Carpeta de Investigación: Análisis de Algoritmos

Título del proyecto: Análisis de Algoritmos

Inga Gonzalo (ingagonzalo1999@gmail.com)

Materia: Programación I

Profesor: Ariel Enferrel

Tutor: Maximiliano Sar Fernández

Fecha de entrega: 20 de junio de 2025

Objetivos:

Fundamentos del Análisis de Algoritmos: A través de explicaciones teóricas y ejercicios prácticos, los estudiantes comprenderán la importancia de evaluar la eficiencia de los algoritmos. Aprenderán a expresar funciones de tiempo T(num) y a identificar patrones de crecimiento en distintos tipos de algoritmos.

Introducción:

El siguiente trabajo se enfocara en profundizar en el análisis de algoritmos, uno de los conceptos fundamentales a la hora de enfrentar un problema, ya que sin su implementación, no podríamos encontrar la forma más eficiente y escalable de solucionar un problema, esto lo abordaremos, partiendo de conceptos fundamentales, como lo es la definición de algoritmo, las diferentes formas de realizar un análisis, sus ventajas y cuando utilizarlas, la idea con este trabajo es enseñar y demostrar su importancia en el mundo de la programación.

En este trabajo se buscó demostrar la importancia del análisis al momento de enfrentar problemas en el desarrollo de soluciones a través de la programación, ya que los diferentes análisis nos permiten evaluación la eficiencia de diferentes soluciones, y

seleccionar la más adecuada en función del contexto, los recursos disponibles y la escalabilidad requerida

Entender estos conceptos y analizar el comportamiento de los algoritmos, se convierte en algo esencial para desarrollar un software optimizado, escalable, y funcional, buscando que este mismo este preparado para manejar gran cantidad de datos, sin poner en riesgo nuestro programa. Con esta base, podemos ver como el análisis de algoritmos, no solo resalta a nivel teórico, sino que resulta ser fundamentar en la práctica diaria en el desarrollo profesional.

El objetivo en este trabajo es profundizar en los conceptos mencionados anteriormente, explorando las distintas formas de análisis que nos permite evaluar la eficiencia de un algoritmo, y demostrando, a través de un caso práctico en Python, como el conocimiento y la aplicación de estos principios impactan en la elección, calidad y rendimiento de una solución.

Marco Teórico:

¿Qué es un algoritmo?

Un algoritmo es un conjunto de instrucciones o pasos, desarrollados de manera precisa y ordenadas que se encargan de resolver un problema en específico.

Los algoritmos deben poseer una estructura definida, debe ser escrito de manera correcta, robusta y eficiente, de manera tal que produzca un correcto funcionar a la hora de someterse a diferentes condiciones, como también, debe ser capaz de administrar los diferentes recursos de manera óptima y razonable.

- **Correcto:** Debe resolver el problema de manera precisa.
- **Robusto:** Debe manejar situaciones inesperadas sin fallar.
- **Eficiente:** Debe utilizar los recursos de manera óptima, especialmente en términos de tiempo de ejecución y uso de memoria.

Todos en nuestra vida aplicamos el uso de algoritmos sin darnos cuenta. Esto lo podemos ver en los aspectos de nuestra vida en donde seguimos diferentes pasos para obtener un resultado final, por ejemplo, al cocinar, al resolver un calculo matemático, o antes de ir a trabajar cuando nos alistamos.

Además de todos estos ejemplos, los algoritmos son un componente fundamental en la generación y funcionamiento de programas informáticos y en el desarrollo de aplicaciones

Se trata de soluciones para automatizar diversas acciones y que, a través de la continua interacción de los usuarios, se perfeccionan día a día y optimizan su fin.

Ejemplo:

```python
def encontrar_mayor(a, b):
    """
    Devuelve el número mayor entre a y b.
    """
    if a > b:
        return a
    else:
        return b

# Entradas
numero1 = int(input("Ingrese el primer número: "))
numero2 = int(input("Ingrese el segundo número: "))

# Proceso y salida
mayor = encontrar_mayor(numero1, numero2)
print("El número mayor es:", mayor)
```

Entrada: El usuario ingresa dos números.

Proceso: Se compara cuál es mayor usando una estructura condicional (if).

Salida: Se muestra en pantalla el mayor de los dos.

Análisis de Algoritmos

¿Por qué es crucial el análisis de algoritmos?

El análisis de algoritmos es esencial para medir la eficiencia de un código en términos de tiempo y espacio. En competencias, donde los límites de tiempo y memoria son estrictos, entender cómo los diferentes enfoques afectan el rendimiento puede ser la diferencia entre aceptar o rechazar una presentación.

1. **Complejidad Temporal:** Evalúa el tiempo necesario para ejecutar un algoritmo a medida que crece el tamaño de la entrada. Por ejemplo, una solución con O(n) generalmente será más eficiente que una con O(n2) en entradas grandes.
2. **Complejidad Espacial:** Además del tiempo, es vital considerar el uso de memoria. Hay momentos en que necesitamos que nuestros algoritmos ocupen menos memoria y necesite menos recursos a la hora de ejecutarse, esto lo podemos ver en situación donde tengamos que trabajar con restricciones de memoria.

Por ejemplo, si tenemos un programa que tarda diez segundos en ejecutarse, nos va a traer muchos mas problemas cuando lo tengamos que usar miles o millones de veces, perderíamos mucho tiempo y recursos empleados solamente en un solo algoritmo, transformando nuestra solución en inviable.

Tipos de análisis:

Análisis Empírico

Esta forma de análisis se enfoca en la implementación de diferentes soluciones en un entorno controlado, y comparar la eficiencia de cada solución de manera que nos permita comparar su eficacia midiendo el tiempo de ejecución.

Pasos para realizar un análisis empírico:

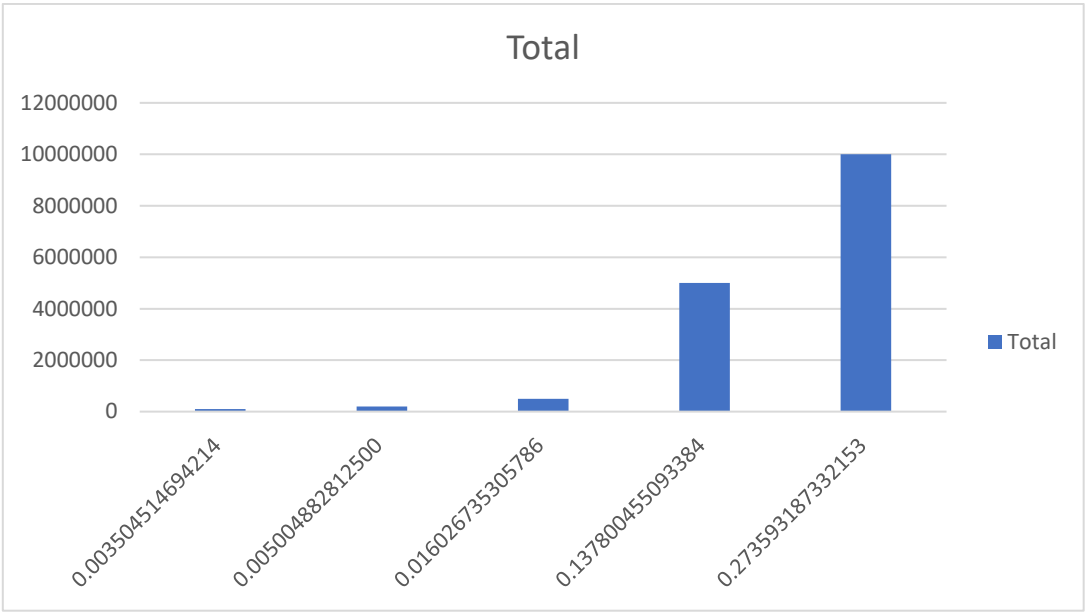
1. **Implementación del algoritmo:** Escribir el código del algoritmo en un lenguaje de programación (en este caso, Python).
2. **Instrumentación:** Incluir instrucciones para medir el tiempo de ejecución, utilizando funciones como time.time() en Python.
3. **Ejecución con diferentes tamaños de entrada:** Ejecutar el programa con entradas de diferentes tamaños y obtener los tiempos de ejecución para cada tamaño.
4. **Visualización de resultados:** Importar los resultados a una hoja de cálculo y crear un gráfico de dispersión (X-Y) para analizar el comportamiento del tiempo de ejecución en función del tamaño de la entrada.

```python
import time

def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

# Prueba empírica
tamaños = [100000, 200000, 500000,5000000,10000000] #Lista de entrada
for tamaño in tamaños:
    lista = list(range(tamaño))
    objetivo = -1  # Usamos esto para que la funcion busqueda_lineal,
recorra todos los tamaños

    inicio = time.time() #Medimos el inicio de la funcion
    busqueda_lineal(lista, objetivo)
    fin = time.time() #Medimos el final de la funcion

    print(f"Tamaño: {tamaño}, Tiempo: {fin - inicio:.15f} segundos")
```

Tamaño	Tiempo(s)
100000	0.003504514694214
200000	0.005004882812500
500000	0.016026735305786
5000000	0.137800455093384
10000000	0.273593187332153

Gráfico de dispersión:

- Eje X: Tamaño de la entrada (n).
- Eje Y: Tiempo de ejecución (segundos).



Ventajas y Desventajas del Análisis Empírico

Ventajas:

- Permite obtener gráficas que muestran el tiempo de ejecución de un algoritmo para diferentes tamaños de entrada.
- Facilita la comparación visual de los tiempos de ejecución de diferentes algoritmos para un mismo problema.

Desventajas:

- Requiere la implementación del algoritmo, lo que consume tiempo y recursos.
- La comparación requiere que los algoritmos se ejecuten en el mismo entorno de hardware y software.
- Los resultados pueden no ser representativos para todas las posibles entradas, ya que solo se evalúan un número finito de ellas.

Conclusión:

- El tiempo de ejecución aumenta con el tamaño de la entrada.
- El análisis empírico nos permite visualizar el comportamiento del algoritmo.

Análisis Teórico

Este análisis nos permite analizar el código, sin tener la necesidad de ejecutarlo, es decir, estudiar al algoritmo y por medio del análisis tener una idea aproximada de su eficiencia.

El análisis teórico es un enfoque matemático que nos permite evaluar la eficiencia de un algoritmo sin necesitad de ejecutarlo. La principal base de este análisis es entender el pseudocódigo del algoritmo, de esta manera se nos permite calcular la función temporal T(n), esto nos representa la cantidad de operaciones que realiza el algoritmo para una entrada tamaño n

Cálculo de la función Temporal T(n)

- **Operaciones primitivas:** son operaciones simples como lo son asignaciones, evaluaciones de expresiones, acceso a arrays, mostrar elementos, entre otros. Todas estas operaciones aumentan en 1 nuestra función Temporal T(n).
- **Estructuras de control:**
    - **Secuencia**: Cuando nuestro algoritmo este compuesto por varios bloques, la suma de las funciones T(n) de cada bloque, concretara el total de la función T(n), es decir: T(n) = T(B1)+T(B2)+…+T(Bn)
    - **Condicionales**: Las estructuras condicionales (if-else) permiten la ejecución de diferentes bloques de código en función de una condición. La función T(n) se define como el máximo de las funciones T(n) de los bloques que se ejecutan.
        - Tif-else (n)= max(TB1 (n), TB2 (n), ..., TBk (n))

      Es decir que tomamos el bloque de código que mas va a tardar en ejecutarse T(n)Max y usamos ese valor en nuestra función temporal.

- **Bucles**: La función T(n) se calculara en base al numero de veces que se va a ejecutar ese bloque multiplicado por la función T(n) del bloque interno.
  - T loop (n) = T(B) * número de iteraciones
- **Bucles anidados**: Producto del número de iteraciones de cada bucle
  - T(n) es: T(n) = n * n * 2 = 2n2

  Es decir, 2 del bucle interno, por el número de iteraciones del bucle interno, por el número de iteraciones del bucle externo.

Ventajas del análisis teórico:

- Podemos realizar este análisis sin depender de nuestro hardware o software
- Podemos analizar todas nuestras posibles entradas.
- Es más general y abstracto que el análisis empírico.

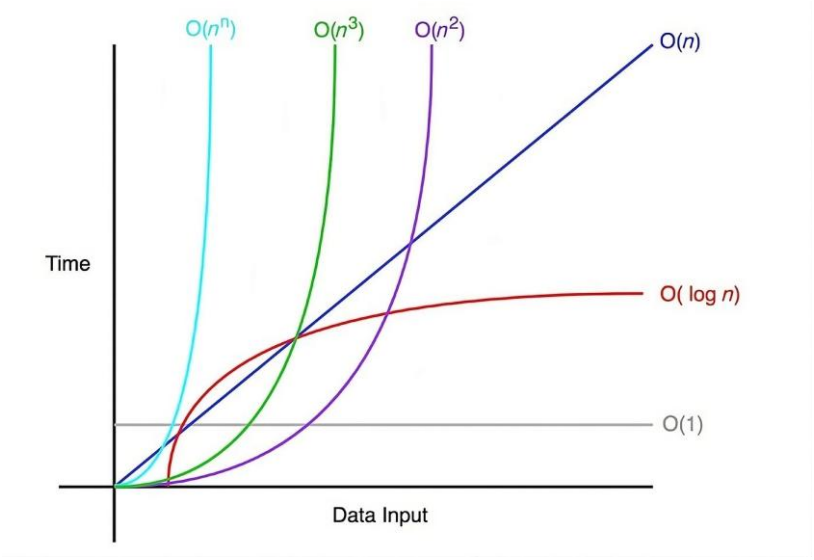
Conclusión

- El análisis teórico nos permite calcular la función temporal T(n) de un algoritmo.
- La notación Big-O simplifica la comparación de algoritmos al enfocarse en el término de mayor crecimiento.
- Es una herramienta esencial para elegir el algoritmo más eficiente en función del tamaño de la entrada.

Notación Big-O

Como observamos anteriormente el cambiar nuestra entrada, por ejemplo, al doble de nuestro primer valor, (100 -> 200), no significa que nuestro tiempo de ejecución de nuestro algoritmo se duplique. Esto depende de nuestro algoritmo, cada algoritmo escala de manera diferente. Para describir este comportamiento utilizamos la notación asintótica.

La notación Big-O es una forma de describir el comportamiento asintótico de una función, es decir, cómo crece cuando el tamaño de la entrada tiende a infinito. Se utiliza para simplificar la comparación de algoritmos eliminando constantes y términos de menor orden



Pasos para calcular Big-O:

1. Identificar el término de mayor crecimiento en T(n).
2. Eliminar constantes y coeficientes.

Cualquier función o línea de código se considera Big-O de 1 o tiempo constante, siempre y cuando no sea un ciclo, no tenga recursión o no sea una llamada a una función que a su vez no sea de tiempo constante

```python
entrada = input()          # 0(1)
x=5                        # 0(1)

if entrada == "holi":      # 0(1)
    print("saludo" * x)    # 0(1)
```

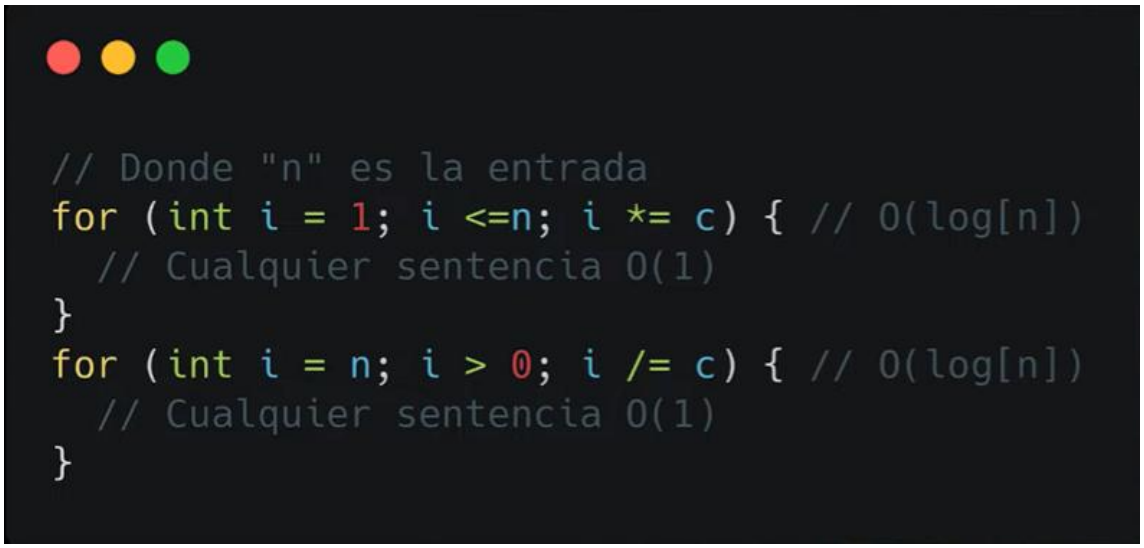
Ciclos: En los ciclos se considera Big-O(n) o de tiempo lineal cuando la variable del ciclo va incrementando o decrementando por un numero constante y siempre y cuando este ciclo vaya iterando en base a la entrada

```java
// Donde "n" es la entrada
for (int i = 1; i <= n; i += c) { // O(n)
  // Cualquier sentencia O(1)
}

for (int i = n; i > 0; i -= c) { // O(n)
  // Cualquier sentencia O(1)
}
```

Ciclos Anidados: Cuando tenemos ciclos anidados, la complejidad depende de cuantas veces se ejecute el ciclo que esta más adentro, dándonos por ejemplo BigO(n^2), cuando son dos ciclos o BigO(n^3) cuando son tres ciclos, y así sucesivamente

```java
// Donde "n" es la entrada
for (int i = 1; i <=n; i += c) {
  for (int j = 1; j <=n; j += c) { // O(n^2)
    // Cualquier sentencia O(1)
  }
}

for (int i = n; i > 0; i -= c) {
  for (int j = i+1; j <=n; j += c) { // O(n^2)
    // Cualquier sentencia O(1)
  }
}
```

En los casos donde la variable del ciclo se multiplica o divide la complejidad se convierte en logarítmica por lo que obtendremos Big-O(Log n)



```
// Donde "n" es la entrada
for (int i = 1; i <=n; i *= c) { // O(log[n])
  // Cualquier sentencia O(1)
}
for (int i = n; i > 0; i /= c) { // O(log[n])
  // Cualquier sentencia O(1)
}
```

Y si la variable del ciclo se va incrementando de forma exponencial entonces obtendremos Big-O(log[log[n]])

Luego de analizar de esta manera todo nuestro código, simplemente sumamos todas nuestras complejidades y lo simplificamos.

Órdenes de Complejidad Comunes

Notación Big-O	**Nombre**	**Descripción**
O(1)	Constante	El tiempo no depende de la entrada.
O(log n)	Logarítmica	Típico en algoritmos de búsqueda binaria.
O(n)	Lineal	Tiempo proporcional al tamaño de la entrada.
O(n log n)	Lineal-logarítmica	Típico en algoritmos de ordenación eficientes.
$O(n^2)$	Cuadrática	Típico en bucles anidados.
$O(2^n)$	Exponencial	Típico en algoritmos de fuerza bruta.
O(n!)	Factorial	Típico en problemas de permutaciones.

Conclusión

- El análisis teórico nos permite calcular la función temporal T(n) de un algoritmo.
- La notación Big-O simplifica la comparación de algoritmos al enfocarse en el término de mayor crecimiento.
- Es una herramienta esencial para elegir el algoritmo más eficiente en función del tamaño de la entrada.

Bibliografía:

https://nic.ar/es/enterate/novedades/que-es-algoritmo

https://tup.sied.utn.edu.ar/pluginfile.php/10244/mod_label/intro/Introduccion%20al%20Analisis%20de%20Algoritmos.pdf

https://tup.sied.utn.edu.ar/pluginfile.php/10244/mod_label/intro/Analisis%20de%20algoritmo%20Empirico.pdf

https://beecrowd.com/es/blog-posts/aprendiendo-analisis-algoritmos-programacion-competitiva/#:~:text=El%20an%C3%A1lisis%20de%20algoritmos%20es,aceptar%20o%20rechazar%20una%20presentaci%C3%B3n.

https://tup.sied.utn.edu.ar/pluginfile.php/10251/mod_label/intro/Analisis-Teorico-de-Algoritmos.pdf

https://tup.sied.utn.edu.ar/pluginfile.php/10251/mod_label/intro/Analisis%20de%20algoritmo%20Teorico%20y%20Big%20O.pdf

https://tup.sied.utn.edu.ar/pluginfile.php/10247/mod_label/intro/Notacion-Big-O.pdf

https://www.youtube.com/watch?v=MyAiCtuhiqQ